Lecture 4: AI Agents & Tool Use

Building an Application Routing Agent

University of Chicago

January 27, 2026

Outline

Outline

What We've Covered So Far

Lecture 1: Foundations

- Understanding LLMs, tokens, context windows, and APIs
- Token economics and pricing

Lecture 2: Building AI Systems

- Vertical slices, Crawl-Walk-Run methodology
- Building a resume scoring system

Lecture 3: Improving Performance

- Decomposition, grounding with citations, few-shot examples
- Improving the resume scorer

Limitations We've Encountered

Our current systems are limited to text input and text output:

- LLMs can only process text and generate text
- They cannot directly:
  - Query databases or retrieve external information
  - Call APIs or interact with web services
  - Schedule meetings or send emails
  - Make decisions that trigger multiple sequential actions
- Each call is independent - no persistence or state management

Today: Learn how to give LLMs the ability to interact with the world through **tools** and **agentic systems**

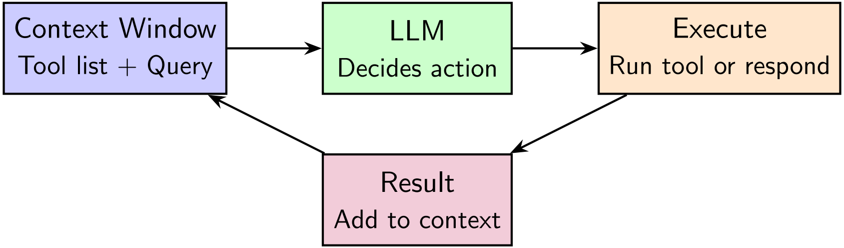
Outline

What is an AI Agent?

- An **AI Agent** is an LLM that can use **tools** to interact with the outside world
- While a basic LLM can only generate text based on its training data, an agent can interact with "tools" that:
  - **Database Tools**: Query databases, insert/update/delete records
  - **API Tools**: Call REST APIs, interact with web services
  - **File System Tools**: Read/write files, list directories
  - **Code Execution Tools**: Run Python code, execute shell commands
  - **Web Tools**: Search the web, scrape websites
  - ...and more

How Do They Work?

Agents are surprisingly simple:



Key idea: The LLM chooses which tool to use (or decides not to use any)

The Agent Loop

We can simplify this into "The Agent Loop"

$$\text{Observe} \rightarrow \text{Think} \rightarrow \text{Act} \rightarrow \text{Observe} \rightarrow \ldots$$

1. **Observe**: Receive input (user query, tool results, system state)
2. **Think**: Process information and decide what to do next
3. **Act**: Execute actions (call tools, generate responses)

Each turn of the loop:

1. Agent sees: current state + what's been done so far
2. Agent decides: which tool to call next (via LLM)
3. Execute tool, log result
4. Repeat until agent calls 'done' or max turns reached

Core Components of an Agent

Every agent system has four main components:

1. **System Prompt**
   - Overall system parameters
2. **Tool Registry**
   - Collection of available functions/tools with their descriptions and required parameters
   - The agent may choose from this registry
3. **Action History / Conversation Memory**
   - What tools have been called so far?
   - What were the results?
4. **User Prompt / Task**
   - The user-provided prompt

The Tool Ecosystem

Different approaches for defining and sharing tools:

- **MCP (Model Context Protocol)**
  - Standardized protocol for connecting AI assistants to external tools and data
  - Developed by Anthropic; allows tools to be shared across different AI systems
  - Example: Database connections, API integrations, file system access
- **agents.txt**
  - Markdown file describing available tools and agent capabilities
  - Simple, human-readable format for tool discovery
  - Often placed in project root for documentation
- **Skills / Plugins**
  - Pre-packaged tool collections for specific tasks
  - Examples: PDF parsing, web scraping, git operations
  - Can be installed and configured per-project

Example: Weather Query (Context)

Agent has access to a web search tool

User asks: "What's the weather tomorrow in Chicago?"

Context window contains:

- **System Prompt:**
  - `Use the tools below to answer the user's query. Return your decision as JSON: {"tool": ..., "parameters": {...}, "reasoning": "..."}"`
- **Tool Registry**:
  - `web_search(query: str) -> str`: Search the web and return results
  - `done(message: str)`: Signal completion with a final message
- **User Query**: "What's the weather tomorrow in Chicago?"

CODE sends this to LLM

Example: Weather Query (Response)

LLM Response (Turn 1):

JSON Output

```
{
  "tool": "web_search",
  "parameters": {
    "query": "Chicago weather tomorrow forecast"
  },
  "reasoning": "Need to search for current weather
                forecast to answer the user's question"
}
```

CODE Response:

1. Parses this JSON
2. Executes `web_search("Chicago weather tomorrow forecast")`
3. Gets result: "Tomorrow: 45 deg F, partly cloudy, 20% chance of rain"
4. Adds result to context for next turn

Example: Weather Query (Turn 2)

Updated context window now includes:

- Original query
- Tool registry
- Previous action: Called `web_search`
- Previous result: "Tomorrow: 45 deg F, partly cloudy, 20% chance of rain"

CODE Sends this to LLM

LLM Response (Turn 2):

JSON Output

```
{
  "tool": "done",
  "parameters": {
    "message": "Tomorrow in Chicago will be 45 deg F with
               partly cloudy skies and a 20% chance of rain."
  },
  "reasoning": "I have the weather information needed
               to answer the user's question"
}
```

CODE Response

- Since the tool is "done", the code knows to stop the loop and return control to the user.

Implementing the Loop in Code

This is just a simple loop!:

Pseudo-code

```
action_history = []
for turn in range(1, max_turns):
    decision = call_llm(context, tools, action_history)     # LLM

    tool_name = decision['tool']
    params = decision['parameters']
    result = execute_tool(tool_name, params) # execute tool

    action_history.append({ 'tool': tool_name, 'result': result}) # update context

    if tool_name == 'done':     # Check if done
        break
```

Outline

Common Failure Modes

Agents can fail in predictable ways:

1. **Infinite Loops / Repeated Actions**
   - Agent calls the same tool repeatedly without progress
   - *Mitigation*: Track action history, detect cycles, set max turns
2. **Tool Hallucination**
   - Agent invents tool names that don't exist in the registry
   - *Mitigation*: Validate tool names, provide clear error messages
3. **Parameter Errors**
   - Agent provides wrong types or missing required parameters
   - *Mitigation*: Schema validation, examples in tool descriptions
4. **Premature Completion**
   - Agent calls 'done' before completing necessary actions
   - *Mitigation*: Clear instructions about required steps, validate state

A viral tool for preventing premature completion

Ralph Wiggum Plugin

Implementation of the Ralph Wiggum technique for iterative, self-referential AI development loops in Claude Code.

What is Ralph?

Ralph is a development methodology based on continuous AI agent loops. As Geoffrey Huntley describes it: **"Ralph is a Bash loop"** - a simple `while true` that repeatedly feeds an AI agent a prompt file, allowing it to iteratively improve its work until completion.

The technique is named after Ralph Wiggum from The Simpsons, embodying the philosophy of persistent iteration despite setbacks.

Core Concept

This plugin implements Ralph using a **Stop hook** that intercepts Claude's exit attempts:

```
# You run ONCE:
/ralph-loop "Your task description" --completion-promise "DONE"

# Then Claude Code automatically:
# 1. Works on the task
# 2. Tries to exit
# 3. Stop hook blocks exit
```

Real-World Considerations

Building production agents requires careful thought:

1. **Safety: How do you prevent runaway agents?**
   - Set maximum turn limits
   - Monitor token usage and costs
   - Implement emergency stop mechanisms
2. **Observability: What monitoring do you need?**
   - Log every tool call and decision
   - Track token usage and costs per agent run
   - Monitor success/failure rates
3. **Reliability: How do you handle failures gracefully?**
   - Validate tool parameters before execution
   - Retry with exponential backoff on transient errors
   - Graceful degradation (fallback to human review)

Ethical Considerations

Automated decision-making requires careful ethical consideration:

1. **Bias and Fairness**
   - Agents **will** perpetuate biases in training data
   - Example: Resume screening may disadvantage certain demographics
2. **Transparency and Explainability**
   - Decisions must be explainable to stakeholders
3. **Accountability**
   - Who is responsible when the agent makes a wrong decision?
   - Legal and reputational risks

Human-in-the-Loop

When should humans override agent decisions?

- **High-stakes decisions**: Firing employees, financial commitments
- **Edge cases**: Unusual situations the agent wasn't trained on
- **Uncertainty**: When the agent's confidence is low
- **Regulatory requirements**: Legal or compliance mandates

Best practice: Design explicit "flag for human review" tools

- Agent can recognize when it's uncertain
- Clear escalation path to human reviewers
- Audit trail of why escalation occurred

Outline

Agentic Resume Screener

An agentic system for routing job applications

- Use agentic principles to evaluate candidates
- Use tools to route candidates to specific outcomes

Components

1. **Tool Registry** - Python functions the agent can call
    - schedule_technical_assessment
    - route_to_department
    - reject_application
    - flag_for_manual_review
    - send_email, done
2. **Agent Loop** - Multi-turn decision making
    - Observe candidate features
    - Decide which tool to call
    - Execute tool and observe result
    - Repeat until done

Tool Example: schedule_technical_assessment

A tool is just a Python function

Python Code

```
def schedule_technical_assessment(
    candidate_id: str,
    assessment_type: str
) -> dict:
    """Schedule a technical assessment."""
    return {
        "status": "success",
        "message": f"Assessment ({assessment_type}) scheduled",
        "scheduled_date": "2024-02-15"
    }
```

Tool Registry

- The code contains a **Tool Registry**, which is a Python dictionary
- We provide this in the context window so that the LLM knows what tools are available
- Each item in the registry contains the following information:
  - Description: "Schedule technical assessment for promising candidate"
  - Parameters: candidate_id (str), assessment_type (str)

Example Agent Flow

Candidate: 7 years experience, strong tech match

Turn 1:

- Observe: 7 years, 85% tech match
- Decide: schedule_technical_assessment
- Execute: Assessment scheduled

Turn 2:

- Observe: Assessment scheduled successfully
- Decide: send_email (template: technical_interview_invite)
- Execute: Email sent

Turn 3:

- Observe: All necessary actions completed
- Decide: done
- Execute: Processing complete

Outline

Today's Workflow

You will build an application routing agent: Open the notebook:

`lecture_4_application_routing_agent.ipynb`

Work through Steps 1-6

Observe the agent loop in action

Analyze your results

Discuss with your team